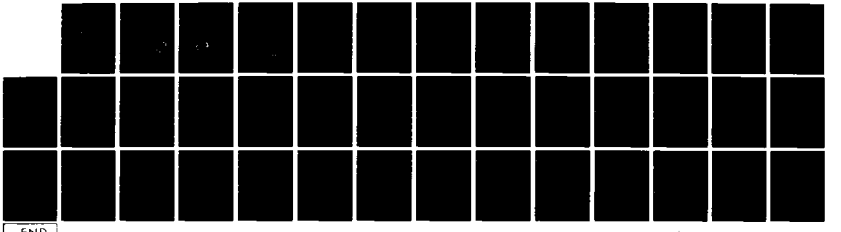
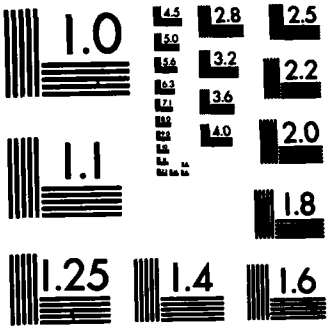
AD-A123 304 LARGE SCALE SOFTWARE SYSTEM DESIGN OF THE AN/TYC-39 - 1/1
STORE AND FORWARD MES..(U) GENERAL DYNAMICS FORT WORTH
TX DATA SYSTEMS DIV 09 NOV 82 DAAK80-81-C-0108
UNCLASSIFIED F/G 17/2 NL



END
FILMED
DTIC



(1)

CASE STUDY I.

# FINAL REPORT DEVELOPED FOR LARGE SCALE SOFTWARE SYSTEM DESIGN OF THE AN/TYC-39 STORE AND FORWARD MESSAGE SWITCH USING THE ADA PROGRAMMING LANGUAGE

U. S. ARMY CECOM
CONTRACT NO. DAAK80-81-C-0108

VOLUME I OF IV


GENERAL DYNAMICS
DATA SYSTEMS DIVISION
CENTRAL CENTER
P. O. BOX 748
FORT WORTH, TX 76101

PAGE

AD A123304

4. Title and Subtitle

Ada Capability Study: Design of the Message Switching System AN/TYC-39 Using the Ada Programming Language

5. Report Date

November 1982

6.

7. Author(s)

General Dynamics

8. Performing Organization Rept. No.

9. Performing Organization Name and Address

General Dynamics
Data Systems Division
Central Center
P. O. Box 748
Fort Worth, TX 76101

10. Project/Task/Work Unit No.

11. Contract(C) or Grant(G) No.

(C) DAAK80-81-C-0108

(G)

12. Sponsoring Organization Name and Address

USA CECOM
Center for Tactical Computer Systems (CENTACS)
ATTN: DRSEL-TCS-ADA-1
Fort Monmouth, NJ 07703

13. Type of Report & Period Covered

Final

14.

15. Supplementary Notes

16. Abstract (Limit: 200 words)

An Ada oriented framework for the design and documentation of the U. S. Army TYC-39 store and forward message switch (military software) system is presented. This document package contains a Requirements, Design, Ada Integrated Methodology, and Final Report section. A methodology to use Ada in specifying requirements, design, and the implementation of a system was developed. This methodology was used to redesign the TYC-39 message switch system. A selected software module was programmed after the redesign.



17. Document Analysis a. Descriptors

Ada Programming Language
Software Design with Ada
Designing with Ada

b. Identifiers/Open-Ended Terms

Message Switch
Military Software
Program Design Language

c. COSATI Field/Group

18. Availability Statement

Distribution limited to the United States. Available from National Technical Information Service, Springfield, VA 22161.

19. Security Class (This Report) UNCLASSIFIED

21. No. of Pages 521

20. Security Class (This Page) UNCLASSIFIED

22. Price

TABLE OF CONTENTS

# 1. Introduction

The Ada Capability Study was performed by the Data Systems Division (DSD), of General Dynamics Corporation, under contract with the Department of the Army Communications - Electronics Command (CECOM). The purpose of this contract was to provide a documented case study and analysis of the use of Ada in the design, development, and implementation of a large scale digital system.

As stated in the Ada Capability Study Work Plan, this task involved the performance of four subtasks: (1) develop a methodology for the use of Ada in the specification of requirements, the design, and the implementation of a digital system; (2) train personnel in the use of the Ada language and the methodology; (3) use the developed methodology to design a system for the AN/TYC-39 message switch; and (4) program one selected module of the designed system.

These tasks have been performed. The Ada Integrated Methodology (AIM) was developed and is provided as one of the documents produced during this study. AIM includes a requirements methodology, a design methodology, and programming standards. AIM has proved to be an effective methodology for the performance of this Ada Capability Study. The scope and applicability of AIM has limitations which are detailed in the AIM document. It is not presented as a general purpose methodology.

The training of personnel was accomplished and a computer assisted instruction sequence was developed. Section 4 of this report provides details.

Using the Requirements Methodology of AIM, an Ada Capability Study Requirements Document was produced which states the AN/TYC-39 message switch requirements in an Ada-compatible format. The Design Methodology of AIM was used to provide a system level design for the entire message switch. A detail design, including hardware-software partitioning, was accomplished for the output message module. Since the purpose of this contract was to study the use of Ada, the design effort was limited in scope, and issues such as hardware producibility, reliability, packaging, EMI, EMC, and survivability were not addressed.

The output message module was coded in Ada. This code was processed through the Ada/Ed interpreter-compiler and is included in the Ada Capability Study Program Listing Document.

## 2. Applicable Documents

The following documents have been produced during the Ada Capability Study contract effort, and are included as a part of the Final Report.

Ada Integrated Methodology, dated 28 June 1982

Ada Capability Study Requirements Document,
dated 9 February 1982, revised 7 June 1982
Ada Capability Study Design Document, dated 29 June 1982

Ada Capability Study Program Listing Document,
dated 21 June 1982

## 3. Project Staffing Review

Ada Capability Study Work Plan was published in August, 1981. The design team lead personnel had been chosen, but the design team staffing was incomplete. It was understood that the success of the contract effort depended to a great extent on the background and qualifications of the persons selected as design team members. The project management therefore sought people with appropriate qualifications and assigned them to the design team as they became available from other projects within the corporation. Highly qualified consultants were also used to support the design team effort.

A total of seven employees were assigned to the design team during the contract effort, with a maximum staff of six at any one time. Two of the people have Master's degrees in computer science; five have Bachelor's degrees, three in mathematics and two in electrical engineering. The leader of the design team was chosen because he had specific experience in communications and telephone switching systems. Other team members had varied backgrounds which included real time systems programming, compiler development, and business data processing. Four of the people had experience in assembly language and Fortran; three had used structured higher order languages including Pascal.

The two consultants who supported this design team effort have PhD degrees and are on the faculty at North Texas State University. They worked in the methodology development phase of the Ada Capability Study and were therefore able to advise the design team members in the application of the methodology.

Because of the varied backgrounds of the design team members, different levels of training were required to ensure the qualification of each person. The training curriculum is described in Section 4 of this report.

Taken as a group, the design team members performed with excellence and did an outstanding job in applying the developed methodology to establish requirements, create a design, and program a selected module using the Ada language in its various forms. Of the seven team members chosen, only one experienced undue difficulty in applying the developed methodology. This person, who had many years of experience with real time assembly language systems, found the methods used to be incompatible with his experience in system development and asked to be assigned to another project.

A willingness to accept new concepts and a positive attitude toward Ada seem to be qualities which were necessary for successful participation in the design team effort. An attitude which reflects these qualities seems to be far more important than the particular educational background or experience of an individual.

It should be noted that the design team was comprised of well-qualified people who were selected for special assignment to the Ada Capability Study and who were highly motivated and intensely interested in the success of the project. This should be considered in estimating productivity for any large-scale Ada system development, because the performance level of a randomly selected group of computer professionals will not be as high.

# 4. Training

## 4.1 Preface

The training of personnel and the development of a coordinated and documented training program have been integral parts of the Ada Capability Study. Early in the program the project management sought effective Ada training in the form of books, video tapes, short courses, and tutorials. It became evident that the availability of training materials at the level required for the Ada Capability Study was limited, and that it was necessary to develop a training curriculum to meet the training requirements of the project. The experience with project in-house training was capitalized upon to produce a formal Ada course in self-instruction format.

## 4.2 Description of Activities

In-house training of Ada project personnel was conducted in two phases. The first phase consisted of ten 2- to 3-hour presentations given to the original members of the Ada project team by Professor Charles Hammons, a project consultant from North Texas State University (NTSU). These presentations were given in seminar fashion, in the form of lectures with accompanying viewgraphs and handouts, and with free class discussion. All features of the Ada language were covered in this phase. Because of time constraints, most topics were covered rather quickly.

The second phase consisted of a reprise of the first phase given primarily for new team members, but open to anyone on the Ada project. These sessions, also conducted by Professor Hammons, covered fundamentals of the language in more detail (in seven 2-hour meetings). Attendees in this phase had, on average, less broad experience in higher order languages than those in the first phase. Special emphasis was given to overall program structure, data types, packaging, and tasking. Phase I experience was useful in identifying and anticipating student difficulties.

All materials used in the training activity will be delivered at the end of the contract period in a separate document.

An important activity in the training section was the formulation and analysis of training issues and training requirements. A related activity was the evaluation of available training materials.

A major effort in the training section was the development of a formal Ada course in computer-assisted-instruction (CAI) format and its implementation on an Apple II personal computer. This self-instructional course can be used by engineering or programming department personnel on-site or on personal equipment. It uses a coursewriting tool written by our NTSU consultants independently of this contract.

A hardcopy version of the CAI material will be delivered at the end of the contract period.

## 4.3 Training issues

A major issue in curriculum development is the question of subsetting the language for training purposes. The experience of the Phase I course showed that thorough coverage of the language in a short time was predictably confusing to students less experienced in higher-order languages.

There are several approaches to subsetting into a core course. Often, such approaches assume an academic context within which not only Ada, but basic programming concepts are taught. This project has assumed that in-house training will be to professionals, and that even entry-level personnel will have the fundamentals of programming and data structures. Approaches to a core course include the following:

- Omit certain topics entirely in the core; for example, separate out system programming concepts such as tasking and reserve for advanced or specialized courses. Problem: in training for embedded systems programmers, it will be difficult to justify omitting "system" programming features.

- Simplify the language by ignoring certain features and options. Problem: it is desirable to cover all features designed to support maintainability and portability.

- Introduce features within a nested sequence of sublanguages. Problem: while this approach is suited to the academic environment, for in-house short courses it loses its designed effect, since the most complex version of language definition is given shortly after the simplest.

## 4.4 Novel Developments

The implementation of a formal Ada course for individual self-instruction on a personal computer is an original contribution to the available body of Ada training materials. Several features of the development tool contribute to the flexibility and modifiability of the course sequence.

It was realized early in the project that effective course materials must reflect a "systems approach" to the language. This means that presentation of language features must be integrated with a view of the system life cycle, and accordingly must be related to program design methodologies, language support issues, and project management tools. The formal course attempts to achieve this integration to at least the minimum extent necessary to make the material appropriate to a variety of potential users.

Project experience in development and application of an Ada methodology provided a unique practical perspective from which to formulate a systems viewpoint. Further, project experience in coding a complex system function provided insight into the training requirements to support such a viewpoint, and into design, program management and coding problems that could be addressed in formal training.

## 4.5 Conclusions

It is difficult to establish a single Ada course to satisfy the needs of industry personnel requiring varying levels of knowledge of the language. Short courses in seminar form plus a self-instructional or programmed learning sequence is suggested as an effective combination.

The most experienced programmers and analysts can profit from an intensive course covering the entire language. Our Phase I experience suggests that it is desirable to have a separate class for this type of student in which potential advantages and risks of the language can be explored in some depth.

Language training will appropriately take place in the context of modern requirements, design, and development methodologies and of the role of program management. Grasping the design of the language requires a perception of how it supports development and maintenance, and of how complexity, portability, and maintainability are managed by production standards. These issues are especially pertinent to the training needs of program managers and system designers.

On the state of language definition: Ada instruction must, in the short term, take account of open implementation and language support questions. These involve significant performance issues such as real time requirements and run time support provided by the Ada system. Students want a clear distinction between what is under programmer control and what is the responsibility of the system.

Class response as well as coding experience on the project has provided insight into some special training issues. Real time, embedded systems programmers need to be shown that Ada will support familiar concepts such as interrupt processing, priority scheduling, critical timing, background/foreground processing, common data pools, and programmer control of execution sequence. Negative transfer from other languages is likely. Some types of lexical errors tended to persist in the coding, for example, failing to specify a discrete range in constrained array declarations.

# 5. Design Issues

## 5.1 Overview

The discussion in this section is the result of the process of designing the TYC-39 message switch. Included are various findings, problems, deficiencies, high points, history, and personal feelings relating to this contract. In addition, the information gained from the various action requests during the project have been incorporated into this material. The actual message switch design is contained in documents accompanying this report, referenced in paragraph 2 (Applicable Documents).

## 5.2 Design Process

### 5.2.1 Requirements Phase

The message switch requirements study began in late July 1981 with a team of requirements analysts composed of the chief engineer, chief programmer, and one assistant who soon became a member of the methodology team. A large part of the early effort was spent perusing the "A level" and "B5 level" specifications and some related documents, namely the Autodin network, JANAP-128, ACP-127, and data adapter specifications. In addition, time was spent researching manual methods of representing and restructuring the data gleaned from these specifications.

On August 4, 1981, the kickoff meeting was held for the representatives of the army at Fort Worth, and the initial plans were revealed. During this meeting, a basic understanding of the message switch was demonstrated by the chief engineer. As is often the case, plans for various personnel were changed and several reassignments occurred. A new chief methodology engineer was appointed, one of the requirements analysts was switched to the methodology team, and three talented consultants from North Texas State University were added.

The chief engineer and chief programmer remained on the requirements team and one new requirements analyst was added. After review of various methods of representing requirements, the SofTech SADT approach was selected because of its ability to separate data and control, two important components of real time systems.

A trip to Fort Monmouth, NJ, was made in early October by the chief engineer, at which time the initial SADT diagrams were presented to the army. After a day-long discussion, some simplification of the overall task was agreed upon, and a better understanding of the message switch battlefield applications was obtained. There were some minor changes made to the SADT diagrams, but the initial understanding was on the right track. Ordinarily, there would be frequent meetings with the customer to resolve misunderstandings and incongruities in the specifications; however, the message switch application is a standard part of the army equipment inventory and the requirements are essentially static. For the purposes of this contract, any conflicts between wording of specifications was evaluated in the requirements group and the most practical approach was taken. This undoubtedly expedited the requirements phase.

By late October the first draft of the Ada Requirements Methodology (ARM) was released from the methodology group. Fortunately, the data flow diagrams were still based on SADT, but additional enhancements were supplied by structured analysis techniques. Some redrawing of the diagrams was

required as a result of the methodology arrival and additional levels of decomposition occurred. Another requirements analyst was then assigned to the project. The chief engineer assigned the analyst to the output message section. The chief programmer was continuing with message routing and the previously-assigned analyst continued on the input message section. Initial assignments were made by the chief engineer based on individual interest, which was somewhat related to their backgrounds. Hardware types worked in areas of I/O and software types in transform analysis. The chief engineer was coordinating the requirements development, assisting in message input, and spending part of his time completing another project unrelated to the message switch (real world problem).

It became evident that the requirements analyst assigned to the message input section was having some difficulty in interpreting the specifications. This could be partly due to the fact that the A level specification contained two pieces of army equipment, only one of which was applicable to the message switch contract. Also, a great amount of implementation detail was specified, which the army had verbally indicated we should ignore. It was desired that the implementation details be driven by the design process (Ada oriented), not the mapping of Ada into the existing design. These considerations added a degree of latitude to the process, but increased the difficulty of getting to the real requirements. Also there was some relunctance on the part of this analyst to conform to ARM (old way is better).

The decomposition process continued into December with the development of a data dictionary of all the data flows on the DFDs, and the lowest level blocks on the DFDs were expressed in an Ada subset as much as possible. Disciplined English was used whenever it was not appropriate to use Ada constructs to express the functional requirements.

The first technical interchange meeting was held at Fort Worth in mid-December. At this meeting, the SofTech group (adjunct contractor) was introduced and a message switch detailed requirements discussion ensued. Initially, it was expected that the design document would be completed by Christmas, but this was not accomplished. The message input requirements were considerably behind, and the requirements analyst assigned at that time asked to be removed from the project. The chief engineer and chief programmer completed the section by late January while the other analyst finished the message output. Also, in January, non-functional requirements were completed, concurrency (high level) was studied, and a concurrency chart was produced.

The efforts of the design team during the requirements phase resulted in a 174-page Ada requirements document. This document restated the "A level" specifications in a more

structured, organized format with many more graphic illustrations of functions.

The application of ARM produced a very good understanding of the problem during the requirements phase. The success in this area can be mainly attributed to the functional decomposition, DFD approach used.

### 5.2.2 Design Phase

A transition from the requirements phase to the design phase took place in late January. The four requirements analysts continued on the project and two new personnel were brought in from engineering. The engineers were given the Ada requirements specification as their primary source of information about the message switch, as well as a briefing on the methodology. The design team met as a group for several weeks, sometimes in full day sessions. Various issues arose during the sessions and the need for individual thought dictated half day breaks on many occasions.

The first two weeks were spent on object-oriented design. Since none of the designers had ever participated in an object-oriented design session, the methodology group was frequently invited. It was suggested by the chief methodology engineer that the search for objects should begin with the top level DFD (node A0). This turned out to be a good idea since four out of seven objects appeared at that level. The process of identifying operations to be performed on the objects gave the design team the opportunity to more closely observe the relationships between data structure components. Thus, information hiding techniques could be applied that allowed operations to be hardware independent. This was especially evident when designing the "reference storage" object, where the "construct" operation was defined to make a sequential operation work acceptably for random access or sequential hardware devices. In addition, the "message" data structure and its components provided the basic structure for the system software design as evidenced by the "message schema" presented in the design document.

During the design sessions, one designer was in charge of updating the chalkboard as the design evolved. The chief engineer refereed the discussions, especially when it was felt that enough time had been spent on a topic. The chalkboard was copied to paper by the participants at the end of each design session or when a new topic was to be considered.

The object-oriented design sessions could have continued longer, but opinions varied as to what additional usefulness would be gained from this relatively new approach. The next step in the methodology lasted about four weeks and began by utilizing traditional structured design techniques to generate a structure chart of the message switch. Although

consideration was given to startup/restart, operator interface, maintenance programs, and runtime support, the primary design emphasis was related to message processing. This was because the message processing is the most important real time aspect of the system and all other software in the switch is present for its support. The support functions were somewhat limited because of the time and scope of the project. The emphasis on support functions was at the interface to the message processing function.

In mid-February a technical interchange meeting was held at SofTech in Boston. At this meeting the chief engineer and the designer who had been transferred from the methodology group presented the results of the requirements phase, object-oriented design sessions, and part of the structured design sessions.

The structured design continued after the interchange meeting. Each portion of the message processing function was being discussed and successively refined. The two engineering personnel made good contributions to the sessions. Between the requirements document and group discussion, they obtained an excellent understanding of the message switch.

After several rounds of refinements, the chief engineer made assignments to team personnel. For those who participated in the requirements phase, the assignments were in a different functional area. This was not only to provide each person with some variety, but also to see if a designer could interpret the requirements written by another requirements analyst. From the time the assignments were made, the person responsible for a particular area of the message switch would be the resident "expert" during a design session involving that area. This helped to ensure that a specific designer was responsible for issues arising during the sessions pertaining to his area of expertise, and part of his time outside the sessions was to be utilized solving these problems.

Throughout February and March, the structure charts were refined, concurrency requirements were identified and coupling and cohesion ("goodness" of design) were evaluated. In mid-March, an in-house preliminary design review was held. All design and methodology team members were present as well as an army representative and consultants. An overview of the message switch was presented for the benefit of the consultants. Then the object-oriented design, structure charts, and concurrency were discussed. Some minor errors were detected during the process and it was generally agreed that the review was worthwhile.

In late March three areas of the message switch were identified as potential candidates for coding as required by the contract. Message output was suggested as the number one

choice and the army subsequently agreed at the mid-April technical interchange meeting. Also in late March three design team members made a presentation regarding the nature of this contract and the progress made to that time at an AdaTEC meeting in Salt Lake City, Utah.

In early April interconnectivity charts were made by each designer for his area of responsibility. The two primary data structures, linetable and routing indicators, were formally organized and recorded. One designer with much hardware experience wrote a description of the "run switch" module in narrative form. This was done to show that an effort had been made to do a complete system design. The area of user interface, startup/restart, fault detection, and maintenance diagnostics are just as important to the system as the application. Due to the size of the message switch and the scope of the project, the level of detail in these areas was necessarily limited. The first seven steps of the design methodology were complete and a one hundred page document was compiled for the critical design review held in mid-April.

All steps in the methodology were useful for design purposes except the interconnectivity charts, which were intended for documentation of interfaces. The information in these charts was derived directly from the structure charts. The CDR was then held at the SofTech office in New Jersey.

After the CDR, the Ada unit specifications for each Ada design unit of the structure chart were created (step nine of the design methodology). The designers accomplished this mostly as an individual effort, with reviews by the chief programmer.

The hardware/software partitioning was done concurrently with the unit specifications. The designer who wrote the "run switch" description worked on partitioning full time. The chief engineer assisted with this process on a part-time basis. In addition to task coordination, time was spent assisting with the data structure unit specs. It was discovered that the distributed processing approach decided upon by the hardware designer could have significant impact on the structure that had been defined up to this point. The level of impact depended on where the partition was drawn. A group meeting was called to discuss the matter, which resulted in a partitioning that had a minimal design impact, yet provided good interprocessor load sharing and cost effectiveness. The conclusion drawn from the experience was that the hardware/software partitioning should be considered earlier, particularly in a distributed environment.

Because of the scope of the project, the detailed design was done only on the selected module and its interfaces. This included the message output section, part of queueing (because of the pre-empt requirement), logging history,

operator malfunction notification, bottom level support for message manipulation and validation, and the user interface relating to system dryup, startup, and shutdown. As the detailed design phase began in mid-April, the first nine steps of the Ada design methodology were complete and the tenth (H/S partitioning) was in process.

The detailed design phase was carried out differently than recommended by the Ada design methodology, partly because the expression of the system design in Ada PDL would not be very different from the requirements RSL that already existed, except in the area of message processing support routines. A two day group meeting was held to establish the exact routines and functions needed for this support, including the memory allocation/deallocation scheme for message buffering. After establishment of this structure, each designer/programmer was to use these support routines as necessary and report to the chief programmer any new support needed but not yet defined. All routines in the MESSAGE_OPS, SEGMENT_OPS, and MANAGE_INTRANSIT packages resulted from these sessions, as well as the message schema, a diagram showing the basic internal message structure. Having these packages at the start of the detail design provided a certain amount of consistency to the resulting design because each designer worked with the same building blocks, not creating individual special purpose routines that partially duplicate functions. This approach worked extremely well. Two designers were paired to design the output message validation routines, another defined the queueing to output port interface, another designed the output port task call/accept structure and support routines, and another continued on hardware/software partitioning.

The final design review called for in the methodology was held at the technical interchange meeting of May 25 and 26 near Ft. Monmouth, N.J. Essentially, there was a complete design walk through (informally held at General Dynamics the week before), a design philosophy review and explanation of the hardware/software partitioning. The requirements-to-design traceability had been completed at the prior design review meeting. Although there was no formal preprogramming Ada evaluation, a set of standards was developed as programming progressed, and groups consulted with each other on a regular basis to ensure that the development stayed on the right track. Since no one had any Ada programming experience, the AdaEd compiler was used frequently to find syntax and compiler errors. This was a valuable tool, even though it was somewhat difficult to use (because of the VAX resources required).

Most of the message switch support routines, queueing interface and validation routines had been written by the late-May technical interchange meeting. The "send message" routines, which required a much closer orientation to the hardware, were written in June. It was quickly determined

that Ada has some deficiencies when interfacing at the hardware level. These are described in other paragraphs of this report. Also in June, time was spent formalizing various documentation, including this report.

The conclusion up to this point is that most of the constructs needed to do a real time system development are available in Ada, but require very careful study to use correctly. The team members who did Ada programming became very proficient in its use, partly with help from other programmers and partly through use of AdaEd. The remaining question at this time is whether or not the final compiled code can run fast enough to actually control a message switch. Hopefully, this will be determined in the near future.

## 5.3 Issues Raised as a Result of Message Switch Design

### 5.3.1 System Design

#### 5.3.1.1 Maintainability vs. Reliability

During the design phase, a potential conflict arose several times between reliability and maintainability. An example is in the output message section where there is a requirement to validate certain header information that was previously validated during message input. Since certain code would be identical, there is good reason to create a package containing this shared code that would be called from both sections of message processing. One member of the group was concerned that this approach violated the reliability requirements because any validation error not detected by the routine in input would likely not detect the same error in the output. His suggestion was to have two separately designed (and maintained) sets of code, one each embodied in the input and output sections. This creates not only a maintainability problem, but in all liklihood does not enhance reliability, because of the increase in complexity. This designer was overruled and the common code was inserted in a single package.

#### 5.3.1.2 Error Handling in Ada

One issue which was raised during the project was the question of how a procedure or function should pass information back to the invoking routine when a problem is encountered. The classic method prior to Ada has been to return a status. This usually takes the form of a boolean or an enumeration variable. The status is then tested and the required action based on the status variable's value is performed. The alternative provided by Ada is to have the called routine raise a programmer defined exception. The invoking routine does not have to explicitly test the exception, but must provide exception handlers at the appropriate place. If the invoking routine wishes to continue with some sort of processing after an error, then a local block will have to be inserted in the invoking procedure to hold the exception handlers. If this is done, the amount of code for the alternative solutions (using or not using exceptions) is very nearly the same. In this case, the only advantage to using exceptions is that a function may be used in some places where a procedure would be needed if explicit status was being returned. For further discussion of this problem, see 5.5.4.3.

#### 5.3.1.3 More Ada Implementation Detail Needed

In several areas, the designers felt that knowledge of the operation of a specific Ada implementation would be necessary. One of these areas concerned dynamic allocation of variables. The Ada reference manual does not specify

whether all dynamic variables will be allocated from a common pool, or from separate pools for each type or each access type. The ability to specify STORAGE'SIZE for a collection implies the latter. This knowledge is needed if the designer is to control memory utilization and overflow (as is required in the message switch).

### 5.3.2 Run Time Support

#### 5.3.2.1 Ada Tasking for Real Time Systems

As expressed at several technical interchange meetings, there is considerable concern over the number of tasks created in a complex real time Ada environment. It is conceivable that hundreds of tasks will be required in the message switch as designed. Although processor units exist that optimize context switching, there is some doubt that the overhead can be minimized to the point that message processing (the application program) will not be adversely affected.

#### 5.3.2.2 Distributed Processing Support

All Ada support provided so far is for a uniprocessor environment. At the time of hardware/software partitioning it was determined that a distributed processing approach was needed to handle the traffic as stated in the A level specification. Unfortunately, no Ada documentation exists that explains how tasking, procedure calls, and access types operate in a distributed processing application. The approach taken on this project was that an interprocessor communications handler would act as an interpreter in each processor, changing intraprocessor commands to a form suitable to interprocessor exchange and then re-interpreted in the next processor in the scope of its internal structure (memory layout, resident tasks, etc.). There are potential problems with this approach mainly because this structure is not very transportable and not as maintainable because more special purpose code exists than there would be if this structure were supported by the Ada run time environment. It is recommended that, as soon as possible, standard interprocessor interface packages be developed to work in the Ada environment. For example, one serial bus interface standard is the MIL-STD-1553B bus. A transparent Ada run time support package for this bus should be developed on a priority basis if real time distributed embedded systems are to be successfully developed in a maintainable manner.

#### 5.3.2.3 Custom Run Time Support

There is a concern that additional run time support will not be identified until late in the system design process, thus causing implementation delays while waiting for the run time support changes to be implemented and tested by the vendor.

### 5.3.3 Ada Language

#### 5.3.3.1 Shared_Variable_Update

The generic procedure SHARED_VARIABLE_UPDATE is not sufficient to provide the functions required for mapped input/output. The problem is that the parameter in the procedure is of mode "in out". In this mode, there is no way for the compiler to determine whether the programmer intended a call to SHARED_VARIABLE_UPDATE as a store or as a load. The designers of this project feel that SHARED_VARIABLE_UPDATE should be replaced by two generics, possibly called SHARED_VARIABLE_STORE and SHARED_VARIABLE_LOAD.

#### 5.3.3.2 Strings

Type string is defined as being indexed by the type natural, which does not allow a null string, while null arrays may be defined. If a record is constructed which contains a string and character count, the count must be of a type other than natural if no characters are contained in the string (null). Thus a conversion must be used in order to index by the count.

#### 5.3.3.3 Context_Dependencies_of_Separate_Subunits

Context dependencies can become rather involved for separate subunits. In many instances subunits will not require the context of the parent, nor is it desirable to make the subunit available on a global basis by including it in a library. Systems programmers and maintainers would benefit if a subunit could be specified as "isolated". The term "isolated" in this instance is the same as "is separate", but without inheriting the context of the parent.

### 5.3.4 Standards,_Training,_and_Experience

#### 5.3.4.1 Naming_Conventions

Conventions and standards need to be set for naming. All names, with the possible exception of loop indices used in very small loops, need to be meaningful. For convenience, abbreviation for long terms may be used, but if they are allowed, they should be standardized. (For a more complete discussion on naming, see SIGPLAN Notices Vol. 17, No. 5, May 1982 for an article by Breck Carter entitled "On Choosing Identifiers".)

A special naming problem is posed by tasks, since the name of the task should be meaningful by itself, as well as when combined with the name of its entries. A perfect example of how not to name a task and an entry is provided in the task FREE_VER, whose single entry is FREE_VERSION. When a call is made to this entry, the call reads FREE_VER.FREE_VERSION. An example of better naming is provided by the task DECOUPLE, whose entry is LOG for a call of DECOUPLE.LOG.

Common sense naming may cause problems. For example, in a package with a set of operations on a message where each routine needs a message as a parameter, then it makes sense to use the same parameter name for the message in each routine. Thus given the following procedures:

```
     procedure READ_FROM_PART
       (MESSAGE : MSGID ; ... ) ;
and
     procedure FIND_RI
       (MESSAGE : MSGID ; ... ) ;
```

If READ_FROM_PART is called from FIND_RI the following would result:

```
     READ_FROM_PART ( MESSAGE =>
       MESSAGE , ... ) ;
```

This may not be a problem; however, it may look strange and potentially confusing to the novice Ada programmer.

#### 5.3.4.2 Use of USE

A standard needs to be set for the use of the USE clause (See the comments under paragraph 5.6) and for qualifying names. Because of his knowledge of the system design, the original programmer is likely to use names without qualification. But the maintainer, who must determine the origin of a name in order to derive its meaning, may have a different perspective. More experience with Ada may be required before a reasonable compromise can be reached.

#### 5.3.4.3 Separate Compilation of Subunits

There is need for a standard relating to the use of separate compilation of subunits. It is a convenience to the programmer during development to be able to work on a subunit in a smaller file which is not being accessed by other programmers. Separate compilation may also be useful during debug to limit the size of recompilations. However, separate compilation poses a problem during maintenance, since a subunit then appears "out of context". One solution to this problem might be to use separate compilation during development, but merge the separate files during the final stages of system integration.

#### 5.3.4.4 Exit Conditions

It is possible to conditionally exit a loop in Ada with two different constructs. The first is the "exit when condition" and the other is by placing an unconditional "exit" statement in an if statement. To maintain uniformity, it may be useful to establish a standard favoring one or the other of these forms. Some of the programmers on this project felt that the "exit when" statement was insufficiently prominent in order

to visually denote its importance as an element of control. These programmers felt that using an if statement, with the subsequent change in indentation, made the exit more visible. Of course, the same effect could be obtained by setting a special standard for the indentation of the exit statement.

#### 5.3.4.5 Overloading

Standards must be set for overloading. The overloading of enumeration literals can be very confusing and should be avoided. Overloading of functions and procedures should be allowed only when the same operation or action is being performed by the overloaded routines. An even more strict approach might be to restrict overloading of routines to those produced by instantiation of the same generic. The cases of overloading in this project were produced in this way. Some apparent overloading of names which are in different scopes should be expected in a large project, and may be tolerated if the scopes are sufficiently separated to remove all possibility of misunderstanding by a maintainer. Also, the hiding of names in an outer scope by names declared in an inner scope is to be strictly avoided. The potential for maintainer confusion in such cases is too high.

#### 5.3.4.6 General Standards

A set of standards was developed for the coding phase of the project. Some Pascal standards were modified for Ada use at the start of coding and further modified as the project progressed. See the AIM document Chapter 4 entitled "Ada Development Standards". Some of the items listed in the standards were added as a result of experience in the coding phase, thus the output message module furnished with this report does not reflect all the standards.

#### 5.3.4.7 Allocation of Hardware Resources

Upon interfacing to a hardware integrated circuit (the Intel 8254), a paradox was encountered. The 8254 circuit has three independent identical counter/timer channels. Two possible choices are:

a. Define the utilization and mode of each of the three channels in a single package, or

b. Define the above in those packages in which a channel (or channels) is used.

The former is advantageous since the hardware utilization is specified in one place. If the latter method is used, one cannot easily determine what portion of the hardware is already allocated, or where it is defined. On the other hand, defining such in a single package does not hide those channels from those compilation units that use only a portion of the resources.

## 5.4 Lessons Learned

Summaries of the lessons learned from the Ada Capability Study are provided in the following paragraphs.

### 5.4.1 Importance of Using a Methodology

Methodologies provide a plan or road map for system development. Without a plan, the system development of any major system is more susceptible to failure. A methodology, even though it may only be a framework such as AIM, forces the project team personnel to think about the problem and solution in an organized manner. In the words of one of our consultants, "Any methodology is ninety percent good".

### 5.4.2 Importance of Understanding the Problem

It is extremely important to develop an understanding of the problem in the requirements phase of system development. This is true regardless of the type of system being developed (i.e., business, real time, scientific, etc.). Once a good understanding of the problem is developed, the design process is ready to begin.

The design team used much more time and effort than originally anticipated to complete the requirements phase. However, the feeling is that the extra time was well spent. The requirements analysts developed an understanding of the problem that reduced the effort required during the design phase.

### 5.4.3 Use of Ada as an RSL Reduces Design and Programming Efforts

The Ada Requirements Specifications produced during the requirements phase eliminated the need for an Ada PDL expression of the system during the design process. The design team felt that the Ada Requirements Specifications were sufficient specifications to begin program development once architectural design was completed. The programming effort was also reduced because the frameworks of many of the Ada procedures were established during the requirements phase.

### 5.4.4 Lack of Integration Between Structured Analysis and Structured Design

Structured Analysis and Structured Design methodologies do not integrate as smoothly from requirements to design for real time communications systems as they do in a business applications environment.

### 5.4.5 Ada Can Be Used Throughout the System Development Life Cycle

Ada has the constructs for forming the base of a structured English for expressing system requirements and programming specifications. Therefore, Ada may be used as an RSL and PDL prior to its use as an implementation language. The use of Ada throughout the system develcpment life cycle reduces the conversion efforts normally required to map requirements intc design and design into ccde.

### 5.4.6 Ada is Most Compatible with the Object-Oriented Design Methodology

Ada will support virtually any methodology. However, it appears that Ada is more compatible with the object-oriented design methodology than other methodologies such as Structured Design, Jackson, and Warnier-Orr.

### 5.4.7 Backgrounds of Personnel Developing Ada Systems is Important

Of course, it is always important to match people with appropriate backgrounds to development projects. The best programmer is not always the best requirements analyst or designer. Therefore, the personnel assigned to the requirements and design phases need not be expert Ada programmers. However, the requirements analysts and designers need to have some knowledge of Ada.

The requirements analysts should be familiar with the basic Ada constructs if Ada is used as an RSL during the requirements phase. Designers should know enough about Ada to use it as a PDL. Additionally, designers should have a good understanding of Ada program structure (i.e., subprograms and packages) and tasking.

### 5.4.8 Need for More Customer-Oriented Requirements Specifications

ARM is heavily oriented toward Structured Analysis. The Ada language is also key to ARM since it is used to express the system requirements. Therefore, the Ada Requirements Document is largely DFDs and Ada Requirements Specifications. This format is great for the requirements analysts and designers. However, from a customer's point of view the Ada Requirements Document is not a good, clear expression of the system. The Structured Analysis and Ada expressions of the system need to be augmented with more customer-oriented system requirements.

### 5.4.9 Value of Graphic Illustrations

DFDs and structure charts are good tools that facilitate developing an understanding of the problem and solution during the requirements and design phases respectively. However, the use of such tools puts Ada in a more supportive role than originally anticipated. The graphic illustrations seem to be easier to understand initially than a pure Ada expression of the problem and solution.

### 5.4.10 Structure Charts are not Designed to Support Recursion

Structure charts have been used prolificly during the design of systems to be implemented in COBOL or FORTRAN. Since neither of these languages is recursive, there is no need for

representing recursion on the structure chart. The lack of a structure chart recursion mechanism may be a problem when trying to model an Ada solution that is recursive.

### 5.4.11 SREM-type Concurrency Charts are not Appropriate for Representing the Concurrency of the Message Switch System

The design team was unable to use the SREM-type concurrency charts to represent the concurrency of the message switch system. One of our consultants tried extensively to draw a concurrency chart for the message switch system but was unsuccessful. A second consultant indicated that it was impossible to illustrate the concurrency of the message switch using SREM-type concurrency charts.

### 5.4.12 Automated Design Aids Could Improve the System Development Process

Automated design aids could be used to improve project management, increase productivity, and improve the accuracy of requirements and design specifications. Specifically, automated design aids could be used to do the following:

- Draw DFDs and structure charts initially or from analyzing Ada code
- Develop traceability matrices
- Structure requirements and design specifications for clarity
- Develop a data dictionary
- Verify requirements and design specifications
- A cross-reference tool to list the location of packages, type definitions, variable declarations, subprogram and task specifications, etc.
- Additional cross reference tools.

## 5.5 Observations

### 5.5.1 Consultants' Comments

- The use of Jackson structure diagrams to depict objects, attributes of objects and operations on objects provided a common communicational tool for the project personnel. The personnel quickly became fluent with this notation and communicated with other project personnel successfully in this environment.

- As one might expect, differences arose as to which components of the system were indeed objects and to what level of detail these objects should be modeled. These differences were generally resolved as the proponents presented arguments for their view of the system and were required to support those arguments.

- The initial attempt at object development was based on the requirements documents. This initial attempt was "reasonably close" to the final version of objects. Thus, the iterative process of refining those initial objects was not protracted. That initial model was an excellent foundation to work from.

- Clearly, some design decisions which promoted information hiding evolved from the object oriented approach. The operations on objects became more clearly defined within the design sessions leading to some general operations, thereby increasing the simplicity of some objects.

- The time spent by the requirements team during the requirements phase definitely facilitated the design process at all steps within the methodology. However, this was very evident during the object development as the requirements team members contributed heavily to the object-oriented model.

- While the knowledge gained during the requirements phase was fundamental to understanding and constructing the object-oriented design, it was felt that there was little formal connection between the two (e.g., traceability).

- Within object-oriented design, concern was expressed over specifying how the objects interacted with one another (this was referred to as the "glue" which tied the objects together). Specifically object-oriented design does not specify flow of control, this is accomplished using the Ada PDL.

- Because flow of control is not specified in the objects, concurrency was not indicated (except through the replication of objects).

- Special difficulty existed in representing a message object, mostly because it "migrated" through the other objects, changing internal representation.

- The fact that the design team was highly trained and had experience with real time processing, seemed to facilitate the use of all methodologies.

5.5.2 Project Members' Comments

- Very few general-purpose packages were developed in the course of this project. These are packages that are suitable for use in a library of such packages; i.e., they are "off the shelf" items. The availability of such packages will greatly reduce coding time over the course of several projects. However, the writing of these packages may take additional time in order to write the necessary extra documentation for future users of such packages.

## 5.6 Values and Drawbacks of Ada

### 5.6.1 General Virtues of an HOL

The use of any high level language (versus the use of assembly language) automatically generates benefits in two areas: an increase in the speed of program development, and an increase in program readability and understandability. Studies have shown that the rate of program development (about ten lines per programmer per day) is largely independent of the language being used. Since a high level language generates the equivalent of many lines of assembly language for each line of HOL, a particular program function can be developed in considerably less time with the HOL than with assembly language. In addition, the use of an HOL frees the programmer from the tedious details which are associated with the use of assembly language, and the many possible errors for which its use creates a potential, such as losing a value through failure to store it, or failing to save the proper registers during calls or interrupts. The readability of a program in an HOL is enhanced by such things as meaningful names, decision constructs, and the very absence of the tedious details mentioned above.

### 5.6.2 General Virtues of a Structured Language

Structured languages have two major benefits for program development which nonstructured languages (including nonstructured HOLs) lack. The first benefit, an increase in the ease of program design and an increase in readability, is generally recognized, and will not be discussed further. The second benefit is the fact that a structured language can be used as a program design and documentation language (PDL). Using the same language for PDL and for coding simplifies training by making the required training in the PDL simply a part of the training in the coding language. In addition, when the design is described in the same language as the program to be coded, the process of turning a design into code is greatly simplified and will proceed much more quickly.

### 5.6.3 Specific Virtues of Ada

#### 5.6.3.1 Strong Typing

The fact that variables of two different types may not be mixed in an expression or assigned to each other without an explicit conversion helps prevent errors in coding, although it does occasionally add apparent complexity to an expression. (No real complexity is added by an explicit expression. The complexity already exists - the conversion just makes it visible.)

#### 5.6.3.2 Tasking

Ada's tasking capabilities provide an excellent way to express any requirements for concurrency which the design may possess. The abilities to have task types and to dynamically allocate new tasks as required can add flexibility to a design.

5.6.3.3 Packaging

The designers of this project found four criteria for building packages, each of which seems to have its place: packaging around a data base, packaging by major program functional area, packaging around a type (or an object), and packaging for general purpose hardware support.

Packaging around a data base is illustrated in this project by such packages as BI_OPS and LINE_TBL_OPS. By placing information used by many modules in a package and restricting access to the data in such a way that the only way the data can be read or written is through the functions which are also contained in the package, the integrity of the data may be more easily ensured. Maintenance is also made easier by the fact that the only access to the data is through the routines in the package.

Packaging by major program functional area is illustrated in this project by such packages as PHYSICAL_PCBT and PROCESS_MESSAGE. In this technique, the package is used as a container for the routines and tasks of some major functional area of the program, along with the types and variables required for their interfaces. The technique would seem to be best suited to projects in which the major areas of the program function independently, and do not have a "driver" controlling their operations.

Packaging around a type is somewhat similar in motivation to packaging around a data base, in that access to objects of the type may be constrained to the routines provided in the package by the use of private and limited private types, but the objects reside outside the package. SEGMENT_OPS and MESSAGE_OPS are examples of this type of package.

During the technical sessions the Army representatives have elaborated on the need for "off the shelf" packages that can be inserted as a particular function is needed. One such package, "Interface to 8254", is a general purpose timer package developed to support an 8254 LSI interval timer. Three Ada language difficiencies make the package less universal than desired. One restriction is more cosmetic in nature and is discussed in paragraph 5.6.5.5. Another relating to hardware interfacing problems is discussed in paragraph 5.6.5.1 and the third problem relating to shared variable update is discussed in paragraph 5.3.3.1. It is fully expected that these problems will recur as other hardware interface packages are developed.

#### 5.6.3.4 Slice Assignments

Slice assignments are a minor but highly appreciated convenience, since they allow the programmer to accomplish in one statement what would require a loop in most languages.

#### 5.6.3.5 Separation of Specification and Body

The Ada capability for separation of the specification and the body of a procedure provided two benefits for the project. The first benefit was that it allowed the interfaces between modules to be written independently of the bodies of the modules, in an early stage of the development of the design. This was especially useful in the case of modules which were used in more than one case, since the specification of such a module could be distributed to all the programmers who were writing modules that called it, so that they would know what syntax to use in the call. A second use of this separation capability was that it prevented circular dependencies from developing between modules.

#### 5.6.3.6 Dynamic Storage Allocation

The dynamic storage allocation capabilities of Ada allowed the designers to make good use of storage without having to divide memory into fixed portions at the start. In order to make full use of this capability, however, the designers would need to know how the specific Ada implementation which they are using accomplishes those functions.

#### 5.6.3.7 Overloading

Project personnel found the overloading capabilities of Ada useful when used for naming subprograms which accomplished the same function on different types. The specific routines with which this was done were FREE and GET. There are three routines by each of these names.

#### 5.6.3.8 Generics

The program designers used generics to develop subprograms such as the above mentioned FREE and GET; the individual versions of which are logically the same.

#### 5.6.3.9 Enumeration Types

The Ada enumeration type is very useful, and more enumeration types were declared in this project than any other user defined type. It proved to be extremely useful to be able to refer to the parts of a message, for example, as HEADER, MSG_BODY, and TRAILER, rather than by number. Other enumeration types were used to represent conditions and error codes.

#### 5.6.3.10 Exceptions

The designers found it quite useful to be able to specify actions to be taken when unanticipated error conditions arose. Predefined Ada exceptions were used in a number of places to handle problems which could have been explicitly checked for by the programmer, and in at least one place an exception was raised by an explicit check. Programmer defined exceptions were not utilized, but they could have been used to pass error conditions up the calling tree.

#### 5.6.3.11 Records

The Ada record type proved to be the second most commonly defined type in this project. Records proved to be valuable in linked lists and other linked data structures, as well as in input/output operations and as parameters.

#### 5.6.3.12 Named_Association

All the personnel employed in design and coding on this project felt that the use of named association in procedure and function calls and in record aggregates made the code much more easily understandable.

### 5.6.4 Dangerous_Features_of_Ada

#### 5.6.4.1 Overloading

The improper use of overloading can create problems in both design and maintenance, since it may be difficult for programmers (both the original coders and the maintainers) to determine what is being referred to, even when the compiler is able to resolve the overloading with no difficulty. The designers on this project feel that overloading should be strictly controlled, and in the case of subprogram overloading, be applied only to those subprograms which accomplish the same action on differing types.

#### 5.6.4.2 Tasking

The Ada tasking feature should only be used by designers who are fully conversant with the dangers of concurrent processing. One very distinct possibility raised by the use of tasking is that of deadlock, also known as deadly embrace. In this condition, two tasks interact in such a way that neither task can proceed without some action on the part of the other. This situation could be difficult to detect in some systems, since all the other tasks in the system might continue to operate normally.

#### 5.6.4.3 Exceptions

The unrestrained use of exceptions to handle problems should be avoided. It is possible for a routine to be aborted

through no fault of its own, and without a chance to "clean up" any of its actions. For example, if routine A calls routine B which calls routine C, and an exception is raised in C which is not handled there, it is possible (especially if the person who wrote B did not know that C could raise this particular exception) that control could revert to A without B having any chance to undo any of its actions. The use of "when others" in the exception portion of a routine also has its problems, the main one being that an exception of an altogether unexpected type may occur, perhaps being raised by some routine which the current routine does not directly call.

### 5.6.5 Problem Areas in Ada

#### 5.6.5.1 Machine-Specific Programming

Although the Ada facilities for machine-dependent programming look adequate at first glance, actual use reveals severe deficiencies. This is especially problematical in embedded systems, where much machine-dependent programming is usually done. For example, the code statement provided by Ada seems to be extremely inflexible. There seems to be no way to use the code statement with a particular Ada variable, so code statements cannot be used for memory-mapped I/O. (Although a sequence of code statements to output any register could be written, there seems to be no way to ensure that what is to be output is in the register.)

The use of address specifications in interrupt handling imposes a problem, due to the requirement that the expression in an address specification be static. This limitation precludes dynamically changing the assignment of tasks to interrupts, which is a requirement of some systems. This problem is discussed at length elsewhere in this report.

Another problem is caused by the requirement for static expressions in addresses. A package that exemplifies this restriction is called "Interface to 8254", which is a general-purpose interface package developed to support an 8254 LSI interval timer. This package is a driver for the given integrated circuit. All communication to this peripheral device is handled by the package, since the address of the chip is known only inside this package. This works acceptably for systems with one such chip. However, in systems with multiple interface chips of the same type, there are two conflicting solutions to the address specifications. One solution is to create multiple copies of the package. Except for the package names and hardware addresses, these packages would be identical. This textual duplication will create maintenance problems. A second possible solution is to create a package that will handle one or more of a given chip. This would probably complicate the single-chip package considerably, since it seems the amount of code would be proportional to the number of chips. For example, the

reading of the three channels of the 8254 were implemented in a case statement, with each selection's code differing only by the names of the variables defined at the respective addresses.

#### 5.6.5.2 Separate Compilation

This problem and the ones that follow it differ from the previous problem in that they are not problems with language capabilities, but with the proper application of these capabilities. While separate compilation may be a solution to one Ada problem (see the next section), its use adds complications to the testing and maintenance portions of the software life cycle. One of these complications is the issue of compilation dependency. On a large project, it may be impossible to keep track of the compilation dependencies. The separation of specification and body into separate compilation units may reduce the amount of compilation dependency. If an environment provides a tool to automatically keep track of dependencies, the problem of deciding whether a particular module needs to be recompiled will remain, even when the tool calls for it, since not all changes to modules on which the current module is dependent will actually cause changes to the context that the dependent module can "see". Another problem stems from the fact that a module which is "separate" inherits the context at the point of its stub. This means that when a maintainer looks at a module, he does not have the full context of that module in front of him. It may actually be in a different file from the one he is editing. In fact, since the containing routine can itself be "separate", the context could be contained in an arbitrarily large number of files. This certainly could make maintenance much more difficult.

#### 5.6.5.3 Nesting of Routines

Ada inherited a problem from Pascal which makes the reading of programs much more difficult. If a subprogram contains nested subprograms, the text of the nested routines appears between the subprogram specification and the body of the subprogram, separating the specification (and the type and object declarations of the program) from the body, sometimes by several pages in large systems. This textual separation can make it very difficult to read and understand the program. One way to solve this problem is to use stubs and separately compile the nested subprograms, but this may create other problems. (See the previous section for a discussion of this problem.)

#### 5.6.5.4 "Use" Clause

The "use" clause is very useful to the programmer during program development. It can save him a great deal of trouble in specifying names in his code. However, the original programmer has an extreme advantage over any maintainer when

it comes to reading and understanding his code, since he knows where his names came from. The maintenance programmer has no such knowledge, and may experience very real problems in resolving the origin of a name in the code. He has only the context and the meaning (if he knows it) of the name to go by in finding out its genesis. Because of this problem, the use of the "use" clause should be restricted to names which come up often in the code. Also, it would be nice if a tool could be developed which would add full qualification to all names in a compilation unit. An alternate tool might provide a list of referenced items for a compilation unit, with their origins.

#### 5.6.5.5 Representation Specifications

The organization of the Ada declarative part with respect to representation specifications (rep specs) presents a problem in readability. All of the designers on this project felt that the required separation of rep specs from their type declarations was difficult to read and would present a maintenance problem. Declaration of a type to be used in communication with a hardware device should appear immediately as needed, not in an apparently unrelated area of the text as is required by the language. One proposed solution would be to declare the types again as a comment in the rep spec area of the declarative part. Although easier to read, a documentation problem will arise in keeping the comment declarations up to date when the real declarations change.

#### 5.6.5.6 Task Rendezvous Mechanism

This mechanism is very versatile; however, when the objective of inter-task communication is to pass data without actually halting either task to wait for the other, there very quickly arises a proliferation of tasks to act as buffers and handle the passing of the data. This condition will vary with the particular application involved but seems very likely to occur in real time embedded applications.

#### 5.6.5.7 Package Size

The packages of the project have tended to become quite large. Some are too large; for example, PHYSICAL_PORT is nearly forty pages long (in sixty-six lines per page format). This size has several drawbacks.

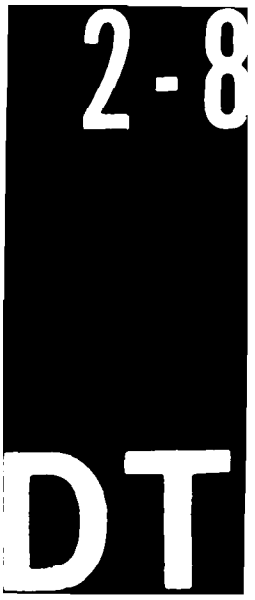
One hindrance is the difficulty of editing a file. The amount of editing of a package is proportional to its size; however, only one person can be working with a given file at once. This causes problems if the package is being coded by several people. Also, one must ensure, before accessing a file, that a person on another terminal is not currently modifying that same file. Furthermore, files with hundreds,

or even thousands, of lines are difficult to edit since the desired section is more difficult to locate.

## 6. Conclusion

The Ada Capability Study has been a success and has demonstrated that Ada can be used effectively in the definition, design, and programming of a large scale digital system. In a span of twelve months, a methodology was developed, personnel were trained, system requirements were defined, a design was accomplished, and a module of the system was coded. Since an Ada compiler and run time support package are not yet available, it was not possible to execute any of the implemented code. It is recognized that certain embedded real time applications may present Ada implementation problems heretofore not realized, particularly in the area of hardware interfacing.

A case study such as this is a good beginning, though it is only the beginning. Continuing research in methodology development and in the use of Ada is required with the development of compilers and an Ada environment.


2-8
DT